NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

REVIEW OF THE PSYCHOLOGICAL ISSUES RELATING TO THE EFFECTIVENESS OF STRUCTURED PROGRAMMING

by

Cynthia A. C. McGrath
December 1984

Thesis Advisor: Gordon Bradley

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Review of the Psychological Issues Relating to the Effectiveness of Structured Programming		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Cynthia A. C. McGrath		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December 1984 13. NUMBER OF PAGES 60
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

cognitive psychology, programming task, understanding task, human information processing system, structured programming

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Claims have been made that the failure of empirical studies to establish the efficacy of structured programming is due to the lack of psychological models of the programming task. Many authors have pointed out that psychological research on the human information processing model might provide substance to the claim that structured programming facilitates a programmer's understanding of program logic. This thesis reviews the results of current psychological research and shows that at (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ABSTRACT (Continued)

this time it is not possible to build a satisfactory psychological model of the programmer and his/her task. In order to define the programming task more clearly, the issues involving the psychological model are identified.

Approved for public release; distribution is unlimited.

Review of the Psychological Issues Relating to the Effectiveness of Structured Programming

by

Cynthia A. C. McGrath
Lieutenant, United States Navy
B.S., Miami University, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December, 1984

ABSTRACT

Claims have been made that the failure of empirical studies to establish the efficacy of structured programming is due to the lack of psychological models of the programming task. Many authors have pcinted out that psychological research on the human information processing model might provide substance to the claim that structured programming facilitates a programmer's understanding of program logic. This thesis reviews the results of current psychological research and shows that at this time it is not possible to build a satisfactory psychological model of the programmer and his/her task. In order tc define the programming task more clearly, the issues involving the psychological model are identified.

TABLE OF CCNTENTS

LIST OF FIGURES

I. INTRODUCTION

A major topic of computer science thinking and research over the past 15 years has been the concept of programming and programming design known as structured programming. Despite its rather extensive treatment in the literature as a basis for improving software quality, conceptual developments have been much more prevalent than the corresponding empirical developments. In fact, the set of empirical studies undertaken on structured programming is in a "sorry state and result from poor theory, poor hypotheses and poor methodology [Ref. 1: p. 401]. " Good theory is a prerequisite to good empirical work--good theory not only contributes significantly to the success of the empirical work, it also helps the empiricist to identify strategic propositions in order to achieve parsimony (the desire to prove, improve, or disprove a theory quickly and with minimal effort). Good theory helps to develop research directed toward understanding as well as prediction [Ref. 1: p. 398].

According to Vessey and Weber [Ref. 1], work on structured programming has tended to follow two streams which they have termed the "characteristics" stream and the "effects" stream. The characteristics stream seeks to show any program can be written using well defined control structures and/or that a program that uses these structures can be proven correct. The effects stream, on the other hand, attempts to model how the use of structured programming might affect the quality of programming practice, such as the readability, clarity, and understandability of the program, greater programmer productivity and reduction of testing difficulties.

Claims are made that the use of structured programming produces these and other cost-effective changes in software practice, yet little progress has been made in building models that support these claims. Rudiments of such theory do exist. Many authors have pointed out that the results of psychological research on the human information processing model might provide some substance to the claim that structured programming facilitates a programmer's productivity.

A. STATE OF CURRENT STUDIES

Vessey and Weber [Ref. 1] and Sheil [Ref. 2] have conducted reviews of the studies done on structured programming. Vessey and Weber have not attempted an exhaustive investigation of the literature. However, they do take an empiricist's view in evaluating a subset of the literature with which they are familiar: namely, the laboratory studies, field studies and surveys undertaken on the effects of structured programming on programming practices. They focused specifically on the practices of program understanding, composition, modification and debugging. In setting the framework for their analysis, they define the programming task as a function of life-cycle versus program activity and suggest that for a more extensive analysis, additional dimensions should be added [Ref. 1: p. 399]. Most of their analysis deals with whether a theory underlies the hypothesis tested and whether the level of abstraction chosen facilitates understanding or prediction. They treat traditional methodological issues in a cursory way. Sheil, on the other hand, evaluates the research done from a methodological point of view. In either case, the reviews are detailed and quite lengthy. They will not be covered here. However, it will be commented that these authors have come to similar conclusions: that the studies done to

substantiate the claims for structured programming have not been well done or well thought out, and that is a major reason that the results do not support the claims. Vessey and Weber [Ref. 1: p. 401] state:

> What we have attempted to do is convince, through example, that the claims made for structured programming are in a sorry state, primarily because the underlying theory on which the hypotheses depend is weak or nonexistent and the choice made of a level of abstraction for the research so far has been inappropriate if understanding is to be obtained.

Sheil deals with behavioral issues as a guide to computer practice as well as to psychological investigation. Sheil [Ref. 2: p. 112] states:

> As a group they are unsatisfactory in that they are methodically weak, the effects they report are small....These failings can, in turn, be traced to an underlying naive view of programming skill which has been shaped more by the fashions of contemporary computing practice than by any reasonable appreciation of the complexity of the behavior.

Sheil notes that since the tests are methodically weak, the results, many of which report negatively on the claim attempted to be proven, are not credible.

It therefore appears that although these authors take a different point of view in their review, they are in the general agreement that the research conducted so far has not produced many worthwhile results. Worthwhile, that is, in terms of supporting the many claims made for structured programming. Here is found the motivation behind this thesis.

B. SCOPE OF THIS THESIS

The intention here is to reiterate that the work done so far has, indeed, been atheoretical and has produced no

definitive results. Further, that although Sheil was partially correct in his view that the programming task has been naively defined, the programming task has also been incorrectly defined. Vessey and Weber, although presenting a good review of the work done, also did not choose the proper perspective from which to view the problem. A better definition of the programming task is necessary, not in terms of the life-cycle as Vessey and Weber chose to do, but, since programming is a human activity, in terms of the human processes involved, e.g., attention, perception, recall, learning, understanding and problem solving.

The intent is to show that with the current state of knowledge in psychology, it will not be possible to build a satisfactory psychological model of the programmer and his/her task. A second objective is to show one can do better than previous work by identifying the components of such a theory and then based on that define the programming task more clearly. In support of this, we investigate the necessary psychological issues on the capabilities and limitations of human memory and the (varied) issues regarding structured programming. It must be emphasized that an in-depth review of these issues is not the intent of this thesis. Rather, only the necessary issues will be expounded upon, while other issues will be dealt with in a cursory way or not at all.

Two questions are of concern: First, given the complexity of these psychological issues, is the research into the effects of structured programming worthwhile to pursue? Second, are measurable benefits derived from structured programming methodology? Even if perceived benefits are real, it is not clear they can be quantified or monitored in order to confirm the effectiveness of the methodologies.

II. THE PSYCHOLOGICAL ISSUES

The programmer brings a variety of tools, experiences and capabilities to bear on the programming task. One of the most important and perhaps the most often overlooked capability is the human mind. The human mind controls how we think. The structure of the human mind determines its capabilities and limitations and our thought processes.

As psychologists have become increasingly interested in issues dealing with the human mind--its thought processes, its organization and, particularly, its capabilities and its limitations, many computer scientists have also begun exploring this area of research. Computer scientists have rightfully looked to this area of psychology in an effort to understand the human ingredient in computer science. In fact, they have delved into it so much that they have been dubbed "armchair psychologists" by Sheil in [Ref. 2]. They have recognized that research in this area is critical if we wish to know how programmers are to apply what mental facility they have to a task at hand in the most effective and efficient manner possible.

The purpose of this chapter is to define and discuss the human aspects that bear upon understanding the programming process: the human information processing model, a clear view of what it means to "understand" a program, a better definition of task and task complexity, problem solving and programming as learned skills, and individual differences among humans. The intent here is to limit the discussion to mature adults (ignoring developmental issues) and focus on the experienced programmer rather than the novice.

A. COGNITIVE PSYCHOLOGY

Definitions of cognitive psychology are broad and may include topics others would place in related fields. The cognitive psychologist studies how people perceive, organize, process and remember information, as well as the different cognitive abilities and how they differ across people. This is quite different from the behavioralist doctrine which for years dominated human experimental psychology and was more frequently associated in computer science with human factors. Human factors has been typically characterized as fitting knobs, displays and workstation layouts to the idiosyncrasies of the human anatomy. Under the behavioralist perspective, the mind, the human information processing system, is treated as an inaccessible "black box" and is essentially ignored and not used to explain relationships between stimuli and responses. However, this same black box is now the primary target of the cognitive psychologist who is primarily interested with how people take in, transform, store and retrieve information.

B. THE HUMAN INFORMATION PROCESSOR

There are many models of the human information processing system, or IPS. Most of these are quite similar. The model to be referred to here is discussed by Tracz [Ref. 3: p. 130-133]. Figure 2.1 is his version of the human information processor. Here, the major components are memory and the processes that control information flow. This conceptual flow assumes a series of processing mechanisms that accept information about the environment, perform general central processing operations, and control motor output.

The organization and limitations on the types of memory components, which will be discussed below, are perhaps the most significant aspect of the human thought process which affects the computer programmer. Other aspects, such as individual differences and expertise, will be discussed later.

C. MEMORY AND ITS PROCESSES

It should be apparent from Figure 2.1 and associated terminology that modern cognitive psychology has been influenced by developments in computer science. There are several types of memories, all differing in their completeness, their duration, and the manner by which material flows in and out of, or between them.

This model of the human IPS consists of four levels of memory: very-short-term memory (VSTM), short-term memory (STM), long-term memory (LTM) and external memory (EM). Other authors such as Newell [Ref. 4] and Norman [Ref. 5], establish the existence of other types of memories or variations to these memories. Therein lies the differences between many IPS models--the definition of memory levels.

The processes which govern the flow of information into, out of and between these memories are the processes of attention, perception, learning, recall and rehearsal. Each emphasizes a different aspect of processing, but all are related. None can be separated from the others. Attention and perception are associated with VSTM while learning, recall and rehearsal are associated with the interactions between STM and LTM. What is known about the capabilities and limitations of these processes will help determine how these processes affect and limit the mental activity involved with programming.

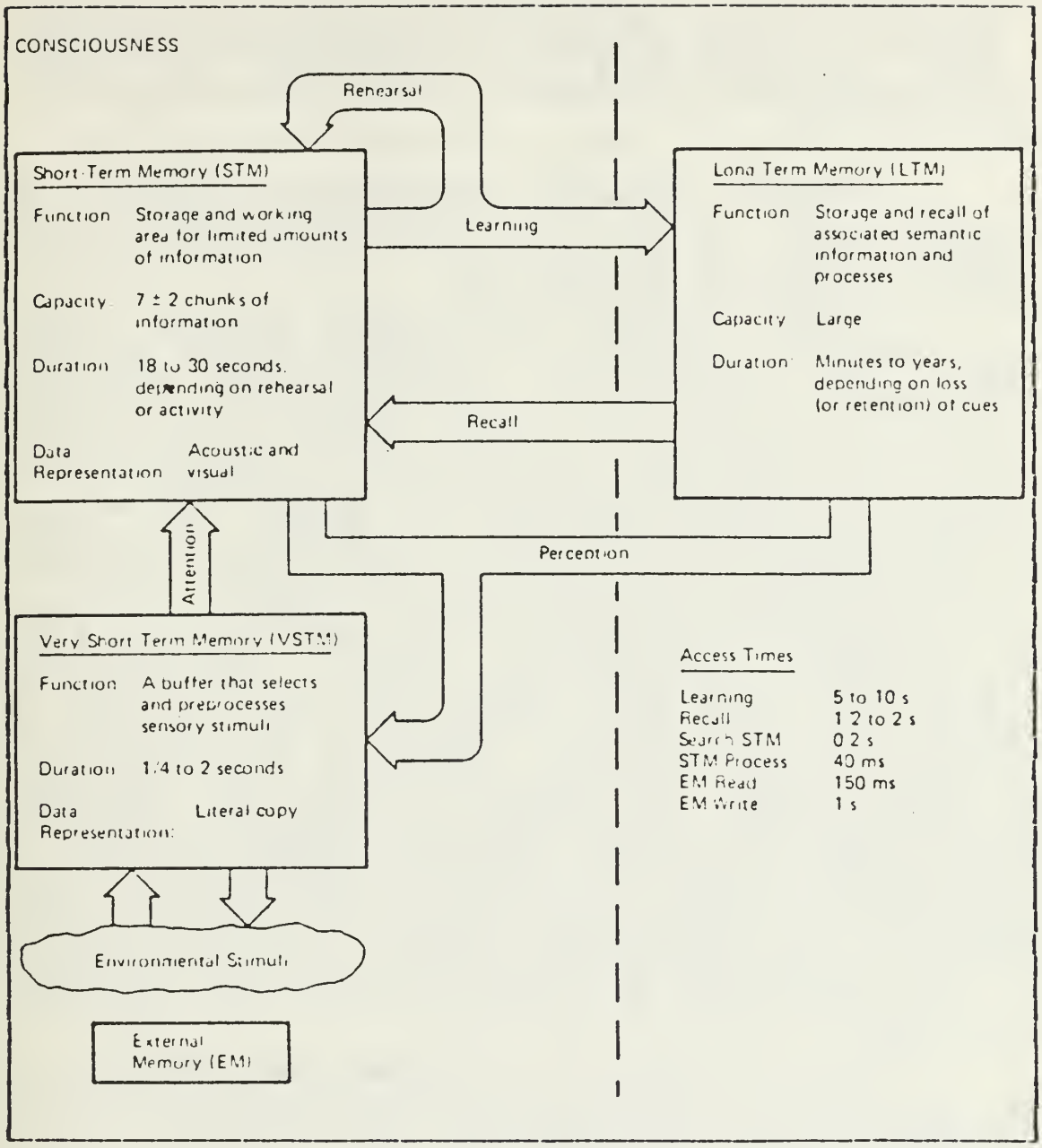

Figure 2.1 The Human Information Processing System.

1. Very-Short-Term Memory

The study of attention is, in part, the study of the limitations on these interacting processes. Attention is defined by William James in [Ref. 5: p. 6] as:

> taking possession by the mind, in clear and vivid form, of one out of what seem several simultaneously possible objects or trains of thought. Focalization, concentration, of consciousness are of its essence. It implies withdrawal from some things in order to deal effectively with others.

Perception (also called recognition) depends on the characteristics of the stimulus and the context in which the characteristics are presented.

Neither attention nor perception operates in isolation. Both require that incoming sensory messages be interpreted with the aid of the context of the messages and their past history. Both context and history are made relevant only through the action of memory. To determine the immediate context of events, a temporary storage system to keep a memory of the recent past is needed.[1]

According to Tracz [Ref. 3], VSTM acts as a buffer for this initial sensory data, holding it for 0.5 to one second with rapid decay thereafter.[2] The process of attention samples the data. The amount of attention given to VSTM controls the amount of information perceived. Through attention, STM can sample the contents of VSTM. The processes of attention and perception, and the mind's interpretation of the information, depends on the characteristics of the stimulus and the context in which the characteristics occur.

This interpretation process has been classified into two major types, differing by how the analysis is guided. In the data driven or bottom-up process, analysis proceeds from the incoming data, through increasingly sophisticated analysis, to the final recognition of the output.

---

[1]To examine the entire past history of an event requires a permanent storage system (LTM).

[2]Access and duration times in the Tracz IPS are based on experiments done on free recall, reaction time, and rote learning.

Whereas this type of analysis does not take into account one's expectations [Ref. 5: p. 41], the second type does.

The second type of analysis is a conceptually driven or top-down process which starts with the conceptualizations of the incoming information. This analysis starts with the highest level of expectation and further refines this by analyzing the context of the stimulus and yielding expectations. This type of analysis can be quite powerful, but its accuracy relies heavily on the selection of expectations [Ref. 5: p. 41]. An inaccurate selection is the major cause for errors in the interpretation of stimulus.

Bottom-up and top-down analyses interact simultaneously. The arrival of information triggers a series of analyses, one of which starts with the components and proceeds to higher processing levels (bottom-up processing). Simultaneously, top-down analysis helps complete the overall 'sense' of the information by taking the context of the stimulus and triggering expectations based on past experience and general knowledge. These expectations produce the top-down processes that eventually merge with the bottom-up processes [Ref. 5: p. 58]. Although both processes are essential and must proceed simultaneously, top-down analysis overrides bottom-up analysis. As already mentioned, this is the major cause of errors [Ref. 3].

2. Short-Term Memory

Recall that the contents of VSTM can be sampled by STM through the process of attention. Rehearsal, on the other hand, is the process by which information is transferred from short- to long-term memory. It is described as a type of inner speech by which we maintain a limited amount of information in memory indefinitely. It is not necessarily conducted using speech mechanisms, but rather, is a simple repetition of items, either silently

and mentally or verbally. Rehearsal is a serial process. Only one item can be rehearsed at any one time. Rehearsal is also a slow process, occuring at the rate of three to six items per second [Ref. 5: p. 100].

As with VSTM, information in STM is highly volatile. Rehearsal maintains, or refreshes an item in STM via this process of silent repetition. Once rehearsal stops, perhaps by switching attention, the item will "fade from consciousness" after 20-30 seconds [Ref. 3: p. 132]. Rehearsal of an item increases its likelihood of being fixed into LTM (learned).

The nature of rehearsal is highly dependent on the nature of the items that are being rehearsed [Ref. 5: p. 100]. For the normal person, when items being rehearsed are words, rehearsal tends to be vocal in nature. When items are not words, but rather sensory in nature (actions, sounds, tastes, smells, visual scenes), then rehearsal tends to mimic the properties of these items. Little is known about rehearsal of nonvocal items.

There have been many studies on rehearsal. From these studies, it appears people perform different operations when rehearsing an item. Maintenance rehearsal is simply a repetitive type rehearsal as described above, and is a natural process. Elaborative rehearsal is described as a "meaningful connections strategy" and characterized by the formation of associations, sentences, images, et cetera [Ref. 5: p. 119]. This type of operation appears to be a learned process.

Miller [Ref. 6] found that STM was limited to holding and processing (7+/-2) pieces of information (commonly called chunks), regardless of the information content of the items. Because of this, Miller found that this apparent limitation could be compensated for by a recoding process known as chunking. Chunking is the

process of grouping information by function and labelling it. According to Miller, the number of pieces of information that STM can contain can be increased by building larger and larger chunks, each chunk containing more information than before.

Learning is the fixation of information into LTM by rehearsal. Like rehearsal, learning is also a slow process, taking five to ten seconds per chunk [Ref. 3: p. 132]. As one might think, the learning time is dependent on the item of information and past experience and knowledge. The learning of familiar pieces of information is faster than the learning of unfamiliar information. Confusion (errors in processing) tend to occur with similar sounding pieces of information.

To recall the past, one needs to retrieve it, recall it back to "conscious" awareness. Consciousness is closely linked to the concepts of attention and to STM, as well as the inner voice within ourselves that appears to analyze our experiences and actions [Ref. 5: p. 217].

3. Long-Term Memory

If there is information to be retrieved from memory, it is useless unless it can be reached. In order to successfully retrieve information, one must be able to determine where it has been stored. This implies organization. How does one search their memory for a fact that is known to be there--somewhere? People tend to group and categorize information they intend to learn [Ref. 5: p. 117]. This can be seen virtually everywhere in society. Telephone numbers are subdivided into small sequences. Children naturally form rhymes of lists they wish to remember. These lists are normally chunked into two or three pieces of information per chunk.

The representation of knowledge in memory is a fundamental and difficult issue to which there is no single answer. There is no evidence that the human LTM is fillable in a lifetime, or that there is a limit on the number of distinguishable symbols it can store. It is generally assumed the IPS has potentially infinite capacity.

Memory retrieval is often a process of problem solving. When a person attempts to retrieve some previously experienced event, the process of retrieval can follow some interesting routes. Problem solving strategies will be addressed in a later section.

Human memory is usually described as associative. Associativity is achieved in the IPS by storing information in LTM in symbol structures. These structures consist of a set of symbols connected by relations [Ref. 4: p. 797]. It is through learning that stimuli or patterns of stimuli become recognizable.

4. External Memory

External memory, like ITM, is essentially infinite in capacity. External memory can be thought of, in the context of this thesis, as the hardcopy listing of a computer program. It is an archive by which exact sequences of statements and subprograms can be retrieved.

EM is not associative, but must be accessed by means ranging from linear scanning to random accessing from addresses built in STM. An IPS with only STM and LTM will behave differently in problem solving than one with EM also.

5. Summary of Memory Processes

One must be wary of comparisons of human psychology with any man-made device. The human mind is not a computer, and the human is not always logical. The differences between the computer and the human mind far

outweigh their similarities. But both do process information. This could cause them to have a number of similar principles, such as the organization of information into meaningful and useful structures.

The answer to memory issues are only partially known today. Unfortunately, much of what is known is still hypothetical in nature. The processes of memory or of learning and recall have not been unravelled sufficiently well to enable all questions to be answered. However, some tentative conclusions can be deduced via these similar principles.

D. PROBLEM SOLVING

Computers were devised tc overcome some of our more obvious limitations, yet their use forces us to develop new skills. As computers solve our old problems, they create new ones. As we expand our capabilities through software development, we are challenged to be more effective in developing programs.

Problem solving can be thought of as a multistage process consisting of: (1) gathering information relevant to the problem, (2) analyzing the data relevant to the problem and proposing solutions, (3) an incubation period during which solutions begin to appear, (4) final synthesis of the solution and (5) verification of the result [Ref. 3: p. 133]. A person uses a variety of methods to analyze the problem. The result of that analysis determines their accuracy in understanding the problem. The primary method to understand the problem is to break down the problem into 'intellectually manageable' pieces to be manipulated be STM.

It is natural to think of problem solving as the accumulation of knowledge. Problem solving depends on the presence of previously learned rules, simpler cases or concepts.

When a problem is first presented it must first be recognized and understood. Then, a problem space must be constructed or if a representation already exists in LTM, it must be invoked. Problem solving takes place by search in the problem space. That is, the process of problem solving considers one knowledge state after another until a desired knowledge state is reached. If the determinant of the problem space and problem are history-dependent, then the problem will change gradually on the basis of experience in problem solving. Therefore, problem spaces can modified during the course of problem solving.

The particular memories and processing rates that characterize humans determine that the problem space is a major invariant of problem solving. (All problem solving occurs in some problem space.) The task environment determines the structure of that space.

In reviewing the works in problem solving, it is important to understand the approaches taken by the authors in the literature in order to evaluate the potential and the limitations of their work.

As a point of introduction to problem solving, Newell and Simon [Ref. 4: p. 3] attempted to compress into one diagram many of the dimensions along which humans vary (See Figure 2.2). Its purpose is not to present a total view. The focus is the individual human being as a system of parts or subsystems. As most authors, they limit their discussion to a few of those parts. The task dimension depicts that humans do a number of different tasks and so behave in a number of different situations, or task environments. The performance-learning-development dimension, which for purposes here will be referred to as the expert-novice dimension, distinguishes among these different activities and correlates them over a time scale to depict maturation. This dimension distinguishes between someone performing a

task, someone learning to perfcrm a task and someone developing with respect to a task. The individual difference dimension distinguishes man as a member of various populations. Differences include age, socioeconomic status, cultures, religion.

1. The Role of Individual Differences

Programming is a mental activity. Programmers differ from each other in many ways. It is not clear that we have assessed all the important mental abilities related to programming. Even with differences commonly noted, e.g. intellectual capability, knowledge base, motivation, personality, and behavior to name a few, the full set of differences which characterize programmer performance has not been modelled and studied under the same data set.

A major belief of most cognitive psychologists is that the basic processes of memcry, attention and cognition are similar for all people. That is, all people have the same form of memory. All people have short-term memory of about the same size. All have the same rehearsal processes and similar representational powers available to them. But, all people are different and their differences are quantitative. One person's short-term memory span might be larger while ancther person's rehearsal rate might be higher.

While the individual differences paradigm provides a method for explaining performance differences in programmers, it offers no explanation of why these differences occur or how to reduce them. Although the individual differences paradigm attempts to assess the mental structure of the human being, it rarely captures the dynamic growth or interaction among those structures. Therefore, its limitation is that it presents a static model of human beings. Since people change over time, a better model is be needed to explain how individual differences occur.

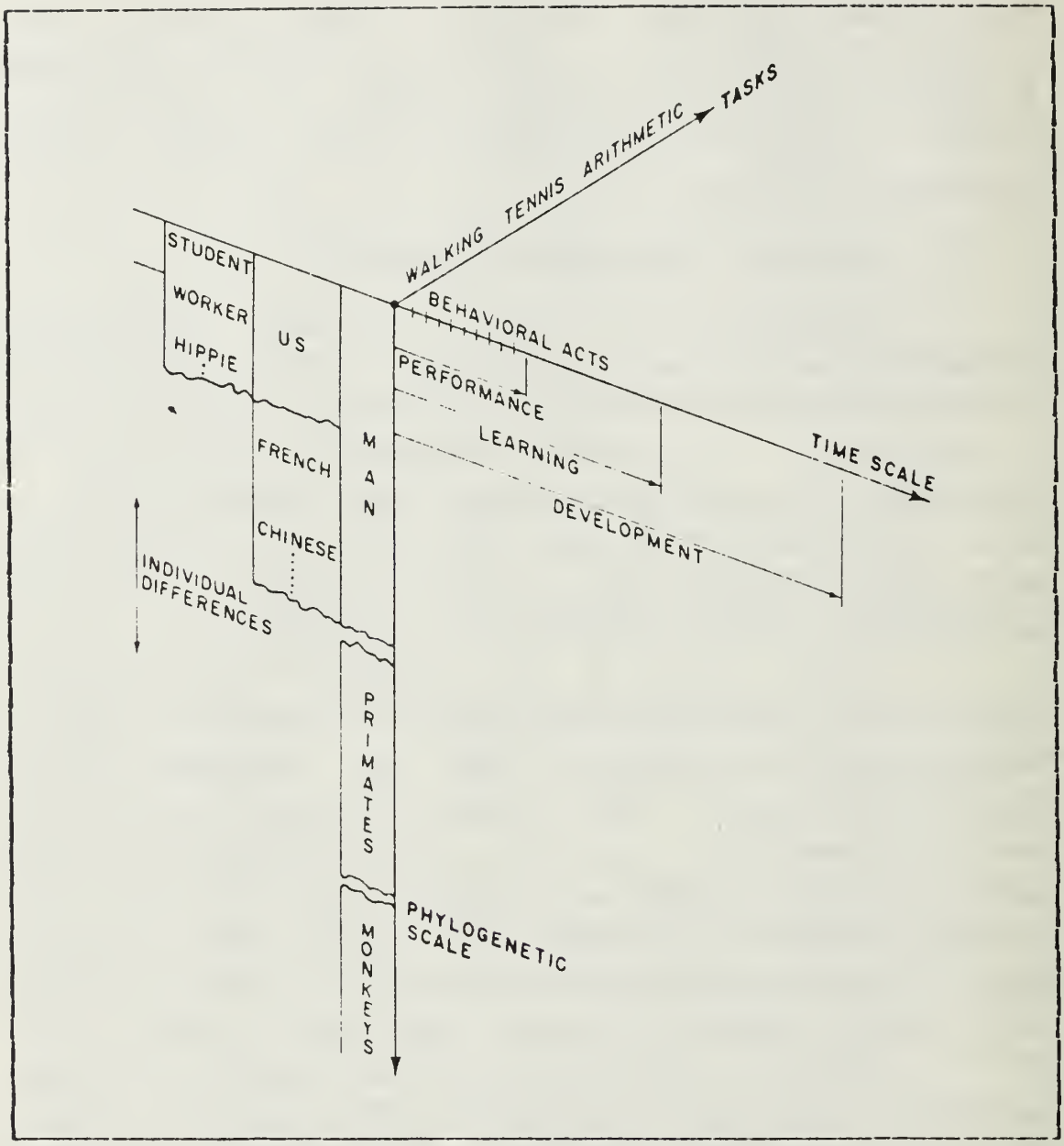

Figure 2.2 Various Dimensions of the Human System.

With respect to programming, there are two major ways in which people might differ. One way concerns mental strategies, particularly with problem solving, and the other concerns the role of prior knowledge and experience. These differences are complementary in that a programmer could

posess a large knowledge base while not having the experience in problem solving to effectively develop and execute the solution to a programming task.

It appears from work in cognitive science that the most important determinant of individual differences in programmer performance is the knowledge base posessed by the programmers. The performance of someone tackling a complicated programming task is related to the extent of their knowledge about that programming area, be it engineering, medicine, physics or any other area.

2. Expert-Novice Dimension

A person's ability as a problem solver is highly dependent on his or her experience. The learning of new information depends critically on what has been previously acquired. We develop expertise in the things we do. One major difference among individuals is their exposure to information. The difference in the knowledge base of the expert over the novice includes a superior facility with specific problem solving methods--the ability to classify problems in a fruitful way, the skill to incorporate new, problem relevant information into memory.

The study of expert-novice differences in programming has generated information on how the programming knowledge base is developed. Results of expert-novice differences in programming determined that experts are not necessarily better at encoding new information than novices. However, the broader knowledge base of experts guides them to quickly cue on the most important aspects of new information, analyze them, and relate them to appropriate schema in long term memory.

It has been demonstrated that novices comprehend a program based on its surface structure (the particular applications area) while experts analyze a program based on

its deep structure (the solution or algorithmic structure of the program.) Further, it was found that knowledge structures developed by experts were more similar to each other than were those of novices, enhancing the ability of experienced programmers to assimilate new information. Another study modelled the programming knowledge base as a collection of plans or templates and demonstrated that programmers can work more effectively when the language they use supports the structure of the templates in their knowledge base [Ref. 7].

There are large differences among individuals in the speed and accuracy with which they accomplish a given task. Theorists in the last decade have devoted increasing attention to analyzing individual differences in the context of information processing models.

People are not all the same. Some may do better at one task than others. Some are good at sports while others are good at painting. Some have an extremely good ability to remember items while others have great difficulty. What are the sources of these distinctions?

STM provides one of the greatest limitations on our ability to develop large scale computer systems. Because of our limits on attention, we are unable to simultaneously keep track on the interwoven processes of a large scale system. Chunking expands the capacity of our STM. Through training and experience, programmers are able to build increasingly large chunks based on solution patterns which occur frequently in problems they solve. On a small scale, for example, an experienced programmer will chunk the calculation of the sum of an array as a single entity rather than a specific sequence of program statements.

Much of a programmer's maturation involves observing more of these patterns and building larger chunks. The particular elements and how they are chunked together

maximize the likelihood of building useful chunks as the knowledge base matures. That is to say, the effects of both expertise and education are on the knowledge base they construct in LTM. The construction is not just the accumulation of facts, but also of organization of those facts into a rich network of semantic material. Experts have more elaborate structures in LTM for encoding designs. They are able to retain to a greater depth since they can use existing structures for reference. New knowledge can be linked to existing knowledge structures then shifted to LTM.

We can see from the above that expertise is specific to knowledge domains. A programmer can be an expert in one domain and a novice in another. The development of expertise involves building a massive knowledge base of recognizable patterns and abstracting a set of rules which govern their behavior. Learning styles play an important role in how quickly, accurately and thoroughly an individual learns.

3. What is Learned?

Simon [Ref. 8] poses the question "What is learned?". When a person learns a task, it is not known what a subject has stored in LTM or in exactly what form he has stored it. Yet the precise nature of what is learned may have considerable influence on both retention over time and the ability to transfer skills to new tasks. Simon contends that different strategies have different degrees of transferability, place different burdens on short term memory and perception, and present different learning processes for their acquisition.

4. Strategies

Problem solving strategies are plans for executing actions in a coordinated way so the solution to the problem is likely to be reached. People vary greatly in how they

choose to represent problems to themselves. The list of strategies are therefore endless. People may be unable to use a particular strategy because it places an excessive demand on their information processing capabilities. People can trade off strengths and weaknesses in information processing capability, general problem solving abilities and expertise in specific areas. Strategies can be used to make up for a loss of basic capabilities (memory deficiency or weaknesses) in solving a problem or learning a task.

Most problem solving research has been done on well defined problems with a finite solution space. In problems such as Towers of Hanoi, used by Simon [Ref. 8], there is an optimal path to the solution. Simon used this problem to show that even in a simple problem environment, numerous distinct solution strategies are available, and different subjects may learn different strategies.

Simon identified four distinct strategies that might represent the knowledge a subject holds after he has learned to solve the Towers of Hanoi problem: rote strategies, goal-recursion strategies, perceptual strategies and move-pattern strategies. The psychological significance of the availability of multiple strategies is severalfold. First, different strategies have different degrees of transferability. Second, different strategies place different burdens on short-term memory, in particular those strategies that depend on a goal stack to hold 'n' chunks of information. Finally, different learning processes may be required for acquiring different strategies [Ref. 8].

a. Strategy for Slicing Programs

Weiser [Ref. 9] introduced the strategy of slicing programs. He defined slicing as "the process of stripping a program of some statements without influencing a given variable at a given statement." Program slicing

is a method used by experienced programmers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program is called a slice. Weiser observed that most programmers unconsciously use this strategy. Limited to code already written, research on slicing may lead to a more complete understanding of the many skills that make up the programming ability and may prove useful during the debugging, testing, and maintenance portions of the software life-cycle and in training programmers.

E. PROGRAMMING TASKS

Webster defines 'task' as "any undertaking or piece of work." The psychological concept of the programming 'task' involves much more. A programming task is an undertaking or piece of work that requires interaction with the programmer's mental processes and knowledge base to arrive at a solution, or the completion of the task. Programming tasks range all the way from learning a language and coding in that language, to debugging, testing and otherwise maintaining the program.

The task environment refers to an environment (or context) coupled with a goal, problem or task. It is the task that defines the point of view about the environment and that allows the environment to be delimited. There is no neutral way to describe the task environment. That is, there is no neutral way to describe the task environment independent of its representation in terms of a particular problem space [Ref. 4: p. 849]. The task environment determines the structure of the problem space.

The task addressed here will focus less on the creation of programs and more on the understanding of programs--a

task of reading and rereading. When a programmer is given an existing program and is 'tasked' with the responsibility of, say, modifying some function within the program, what mental facilities must the programmer call upon? Further, what does our knowledge of the organization and capabilities of normal human memory say about the characteristic of understanding code in the most efficient manner possible?

The scenario of any task can be thought of as a hierarchy of tasks, the completion of the lower level tasks serving as a criteria to fulfill the task at the next higher level. For example, the decomposition of the modifying task can be depicted as in Figure 2.3. Our interest lies strictly on the understanding branch of this tree.

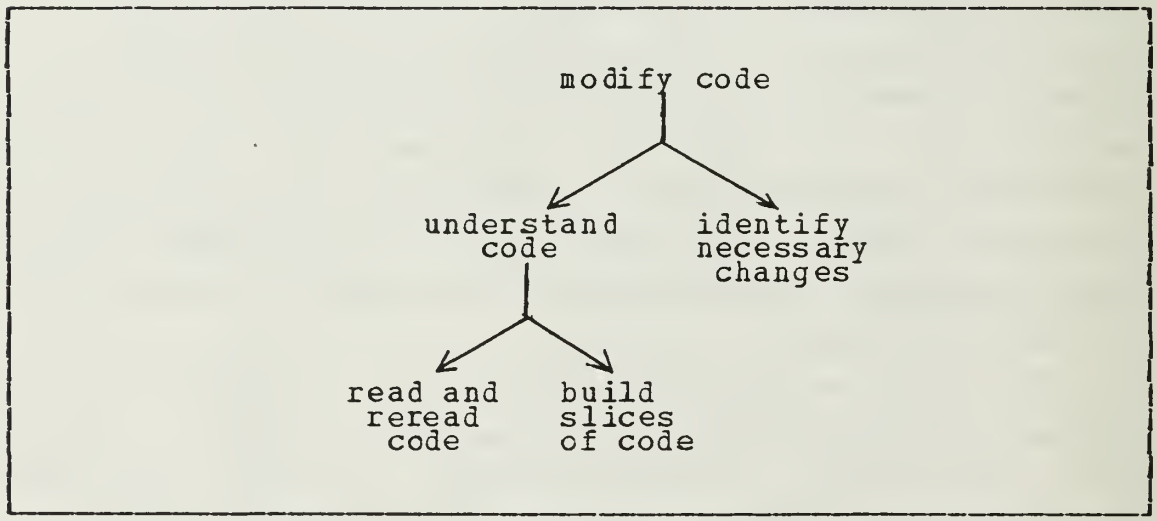

Figure 2.3 Decomposition of the Modifying Task.

1. Reading and Rereading

First, what mental tasks are involved in reading and rereading existing code that a programmer may

have written or may have never seen before? Let us assume the complexity of the program is sufficiently high. That is to say that the length of the program is such that even if the programmer had actually written the program but had not seen the program for a sufficient length of time, the same processes would have tc be dealt with as with the programmer who is seeing the code for the first time. As already discussed, the advantage of the original coder is that as the code is read and reread, certain pieces of information already stored in LTM may be recalled and reorganized, bringing along with it forgotten details. In this way, the original coder is able to move through the understanding processes more quickly than the first time reader.

With this, then, the reading/rereading task at hand could be described as:

1. read the program one or more times for general understanding of program function and organization
2. accumulate (learn) with each repetition some fact(s) about the program, building slices of program organization

What happens with rereading? Once someone learns something, there is something more to be gained by rereading. To understand the program thoroughly, not just the basic principle, but in all its ramifications the details must be thought through. Once the second level is reached, each repetition of rereading produces a more detailed understanding of how the program works. It is at this point that there is sufficient understanding to apply the accumulated slicing knowledge to the task of debugging or modifying the program.

2. Understanding the Process of a Program

Understanding, in philosophical terms, is "the power to render experience intelligible by bringing perceived particulars under appropriate concepts." Programming, like problem solving, is a multistage process. Understanding the problem is the first stage. Understanding the problem can be applied to two situations: generating a new program and modifying an old one. Note that the latter often becomes the former.

Recall that human memory is associative. The process of perception plays an important role in both learning and problem solving. Learning is the fixation of an item into LTM by rehearsal. Understanding additionally involves the association of the concept at hand with stored information (information already learned) and the modification and/or reorganization of that stored information in order to match its associated representation. This reorganization is the result of the recognition of the relationship between two representations of the same thing--the one stored and the one perceived. This understanding task always involves a comparison between two representations. If a second representation is not available, it must be created. If the programming process is difficult to understand (from one programmer's point of view) then it is difficult for that programmer to discover and/or create the relationship between the concept and the alternate representation, or it is difficult to build the second representation. The understanding task is then said to be complex for that programmer. What is complex for one programmer, however, may or may not be complex for another programmer.

a. Understanding in Software Complexity

Two different approaches in relation to software complexity have emerged: computational complexity and psychological complexity. Computational complexity has been fairly well defined and is not of interest here. Psychological complexity, however, is important here because of our interest in how software characteristics affect program understanding. Curtis [Ref. 7] proposes that the unifying thrust for the field of complexity research will be best achieved not by treating complexity as a tangible, readily measurable characteristic of software. Rather, that complexity is an abstract concept which allows us to organize our attack on the problem. He suggests the following definition:

> Complexity is a characteristic of the software interface which influences the resources another system will expend while interacting with the software.

Here, the focus is not just on the software, but its interactions with other systems, such as people. Complexity is defined as a property of the software interface that affects the interactions between the software and another system.

If a program is readable, then it has a textual structure, or format, that gives the reader easy perceptual clues on the details of the program. The limitations of STM mandate the program must be readable to develop the program understanding. To overcome the limitations of STM, the organization of the program should be such that a complete description is presented in a sequential, unambiguous manner. Being complete and concise adds to the manageability of the information and facilitates chunking the components. It also reduces the overall complexity

associated with trying to follow the dynamic structure of a program.

b. Commenting: An Aid to Understanding

When a programmer creates a program, he/she has a wealth of semantic information associated with each chunk he or she manipulates. Unless this information is preserved through commenting (EM), the understander may not be able to reconstruct the subtleties of the program. A program should also complement the human thought process. Abstraction, the process of gathering detail into a workable unit (chunk) in STM, compensates for the limitations of STM and LTM.

The task of understanding the process of a program involves a trace on both specific data and on a class of data, much like the slicing strategies discussed by Weiser in [Ref. 9] and [Ref. 10]. It also suggests a context, a task environment. Structured programming suggests this environment.

III. STRUCTURED PROGRAMMING

A. MEASURING THE "GOODNESS" OF A PROGRAM

Good program design is decomposing in a good way, but what is "good"? Little is known about the goodness of programs and programmers that can be quantified and measured. Most of what is known is in terms of guidelines and philosophies. A good program not only works, but works according to specifications. A good program is flexible and has bugs (which is inevitable) that can be fixed quickly. A good program is well-documented, executes quickly and makes efficient use of memory. A good program is understandable even to programmers other than its creators. A program is good if there exists a "simple" relationship between its specification and its code. Here, "simple" is in human terms. This recognizes the limitations of human processing.

As will be seen, structured programming provides a programmer-independent hierarchical decomposition of programs which relieves much of the difficulty in understanding another person's code. Because programmers develop their own programming style, which is often disorganized, structured programming provides a facility to avoid random programming and so offers a way to narrow the gap in style between programmers. Simplicity in programming style is important to the understanding task. Structured programming suggests a simpler (more understandable) representation of a program's process.

B. EARLY WORK

One of the driving forces in the movement towards structured programming has been Professor Edsger W. Dijkstra. In 1965 he suggested to the IFIP Congress that the GO-TO could be eliminated from programming languages and that 'the quality of the programmer is inversely proportional to the number of GO-TO statements in his program.' Despite the fact that this conference was well attended, the impact of his statement was at the time, small. People had just gotten comfortable with Fortran II and were getting ready for the release of Fortran IV, a language embracing the GO-TO statement. There were also other languages, like COBOL, that programmers had gotten used to and liked.

Dijkstra repeated his ideas in a now famous letter to the editor of the Communications of the ACM in March 1968 titled "GO-TO Statement Considered Harmful" [Ref. 11]. This letter stated that the GO-TO should be abolished from all high-level languages. Dijkstra contended that the difficulty involved in understanding programs which made heavy use of the GO-TO statement was the conceptual gap between the static structure of a program (spread over pages of printout) and the dynamic structure of the corresponding computations (spread over time.) This has since been called the structure principle: The static structure of a program should correspond in a simple way to the dynamic structure of the corresponding computations [Ref. 12].

Later in 1968, Dijkstra again emphasized top-down design, a design that seemed to go hand in hand with structured programming. By 1972 he grabbed the attention of the computer world in what was fondly known as the 'GO-TO controversy.' This led to the loose body of program methods and techniques known as structured programming and to a greater awareness of the need for reliable programming.

By the early 1970's many researchers were interested in these problems, but they chose to direct more of their interests toward programming languages and formal computability theory than to applications of structured programming. Some pointed out that the GO-TO could be eliminated from ALGOL-60 while others developed specialized programming languages and programming styles that eliminated the GO-TO. IBM conducted an experiment where structured programming and top-down design played a large role [Ref. 13].

Dijkstra's objectives seemed relatively constant--questioning whether it was conceivable to increase programming ability by an order of magnitude and what techniques (mental, mechanical, organizational) could be applied to achieve this objective. Dijkstra was also interested in proving the correctness of computer programs. He recognized that this would help eliminate costly, tedious and largely unsuccessful ad hoc testing methods. He did not focus, however, on questions like 'how do we prove a program correct?' but rather 'for what program structures can we give correctness proofs without undue labor, even when programs are large?' and 'for a given task, how can we make such a well-structured program?'

Dijkstra's ideas were simple, but profound. He is noted for having summarized the following characteristics about humans limitations: (1) the human's inability to handle dynamic processes, (2) the human need for a simple correspondence between the static program and the dynamic process and (3) the limited human brain which required a 'separation of concerns.'

C. DEFINITION

Much has been written about the variety of methodologies for computer software development, most of which are founded

on sound, logical principles. In particular, software designers and programmers having practiced structured programming, have asserted qualitatively that they got the job done faster, made fewer errors or produced a better product. Unfortunately, as we have seen, solid empirical evidence that assesses the benefits of structured programming is scarce.

A definition for structured programming is also scarce, and seemingly nonexistent. Structured Programming is not simply 'GO-TO-less' programming, an overly simplistic view that early researchers and others have taken. Structured programming is a systematic way of designing programs, of which eliminating or restricting the use of the GO-TO statement is but a single attribute.

Early on Dijkstra described the principle behind structured programming as: the static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations. This structure is defined by a set of programming rules and restrictions that force the program to follow this tight form. These rules and restrictions eliminate much of the randomness, poor readability, and complexity that leads to bugs and increased testing and maintenance problems.

It is worthwhile to note here, that just as people have different perspectives on the psychological issues involved with the human IPS, so they have different concepts of structured programming. Structured programming has been frequently thought of as (1) a programming style, (2) a programming methodology, and/or (3) a programming technique. Included in style is pretty printing and commenting. Style is observable but can be done mechanically and in a way that yields no improvement. Viewing structured programming as a methodology results in too broad a view. There is no opportunity to view structured programming in an unbiased way

since there is too much to observe and measure. Technique, on the other hand, offers observable properties. It is from this pcint of view we will discuss structured programming: the technique of writing programs according to a set of rigid rules, using well-structured branching and control constructs.

Other conventions include the notion of top-down design, modularization, the concept of levels of abstraction, and lesser important programming restrictions and conventions. Software development, including structured programming, is still very artistic and spontaneous in style, and this is perhaps why an exact definition is difficult to pin down.

D. OBJECTIVES AND MOTIVATIONS

Objectives and mctivations for structured programming should be obvious by now. The primary objective, as always, is to develop minimum cost systems, systems that are as inexpensive as possible to develop, operate, maintain and modify. Scme observers have stated that all things considered, structured programming accomplishes this objective since introducing any degree cf structure makes a more complex system less complex by some factor. If a program is less complex, it is more readable, more manageable and easier to test and debug than an equivalent unstructured program.

1. Clarity and Readability of Programs

An obvious benefit of structured programming, but worth mentioning again, is increased readability of the program. If a program is readable, then it has a textual structure, or format, that gives easier perceptual clues on the details of the program. The behavior of unstructured programs, particularly larger programs containing several

thousand lines of code, are psychologically too complicated, by nature of their unorderliness, for the human brain to keep organized. Any attempt to read a listing where the program executes a few statements, jumps to a point pages away, executes a few more statements, jumps again perhaps in a different direction and so on, is intellectually challenging and very difficult for the normal human brain to remember.

A structured program, on the other hand, is more likely to proceed in a straight-line fashion and is most often accompanied by various formatting (pretty printing, for example) and modularity conventions. This makes the program more organized, appear less complex, and ultimately, more readable.

2. Fewer Testing Problems

Increased readability generally leads to fewer test problems. The review of the literature on structured programming alone, from which a multitude of papers are found, will attest to the fact that researchers largely believe the original motivation of Dijkstra's early work remains the most critical today. The problems of testing are well known. Dijkstra pointed out that testing shows the presence of bugs, but never their absence. It seems bugs remain forever in large programs, particularly those that require continual maintenance, improvements or other modifications. The effort and cost of testing large programs rise exponentially with program size. And with the cost of maintenance experimentally determined to be up to 80 percent of life-cycle costs, it is small wonder so many researchers have devoted so much time and energy on this aspect of structured programming.

One of Dijkstra's objectives in the development of structured programming was that mechanical proofs might be

much easier for a program written with some structure versus no structure. Many others since then have come to feel salvation lies in mechanically proving a program correct. Others suggest testing techniques to help reduce the randomness and lack of organization that accompany many testing efforts. Still, all of these test procedures remain ad hoc in nature. There are many reasons for this. Exhaustive testing for a program of any appreciable size is not only costly, but often impossible. Tests considered often reflect the tester's knowledge of the system or simply the tester's subjective, but perhaps knowledgeable, decisions on choosing a good test case.

The use of structured programming, by way of its hierarchical structure, lends itself to testing modules separately from other modules at the same or different levels. This aids in testing and reduces, at least, some of the "ad hoc-ness" of current testing methods.

3. Increased Programmer Productivity

It is also claimed that reducing testing and associated problems generally leads to greater programmer productivity in that the programmer can generate more debugged statements using the structured programming approach than the unstructured approach. Backing up in the life-cycle, it has also been suggested that each programmer, when using the structured programming approach, is twice as productive in writing code. Brooks [Ref. 14: p. 16] reminds us that "men and months are interchangeable commodities only when a task can be partitioned among workers with no communication among them." This means that although these programmers are twice as productive, less than half of the programmers are required to complete a project, freeing the remaining programmers to take on other projects.

4. Efficiency

Many complain there are negative aspects to structured programming: too many rules and restrictions, eliminating the GO-TO is not good, structure takes away from creativity, less program efficiency, to name a few.

One of the most common complaints against structured programming, primarily among the systems programmers, is that it leads to less efficient programs. Subroutine calls replace the use of the GO-TO statement. The increased emphasis on subroutine calls increases the CPU time of a program. In addition, it could add significantly to the memory requirements of the program.

While there may be some element of validity in the complaint of inefficiency for some types of programmers (like systems programmers), productivity remains but one of the many (claimed) benefits of structured programming. Structured programming produces savings in productivity which many others feel far outweigh the inefficiency inherent in the structured programming approach.

E. THEORY OF STRUCTURED PROGRAMMING

Design is concerned with devising artifacts about goals [Ref. 15]. All problems, particularly large and complex ones, have a large number of possible design solutions. Design goes from the general to the specific statement of the problem by making a succession of design decisions. The choices are based on the desired goals: maintainability, efficiency and correctness are just a few. These goals are conflicting. There are many tradeoffs among goals when settling each design decision.

As we have seen, structured programming design is often considered a philosophy of writing programs with the primary technique being the use of well-structured branching and

control statements. Formal systems of computability theory do not require the notion of the GO-TO. The GO-TO statement, as Dijkstra pointed out long ago, is not needed to compute all functions but many feel it is convenient.

A number of researchers have been attacking the practical goal of writing programs (specifically large and complex ones) in such a way that their correctness can be proven. For the reasons pointed out by Dijkstra in his early work, their efforts have been directed towards programs that are developed in a top-down or hierarchical fashion.

To understand this one must realize the program is never a goal in itself. The purpose of a program is to evoke computations and the purpose of computations is to establish a desired effect. Recall the structure principle, the motivation for which was to narrow the conceptual gap between the static program (timeless object) and the dynamic computation (makes sense only via execution). The goal for writing programs so that their correctness can be proven is to use our understanding of each program and make assertions about the ensuing computations. The ease and reliability in doing this depends on the simplicity of the relationship [Ref. 11] [Ref. 16].

A natural way to simplify the relationship is to develop the program in a top-down fashion. Each design decision has a context. The designer can control the size and complexity of the context by a number of techniques. Abstraction and decomposition are just two techniques.

1. Abstraction and Decomposition

In this scheme, an entire program is considered to be an independent "callable" module. At the next stage of design, the original program is broken down into

subordinate modules. Each of these submodules are then decompcsed further into submcdules. The decomposition continues until the only things left are the building blocks that are small enough to code easily.

In order to test the entire program, it is important to be able to define the behavior of the submodule at the k-th level independently of the context in which it occurs. The correctness of the submodules at the (k+1)st level can then be proven independently of their ccntext in the k-th step. This suggests that each submodule should be designed with a single entry point and a single exit, and in turn, the entire program can by described as a set of nested modules, each of which has one entry and one exit [Ref. 16].

The aim is, in a rough sense, to understand a program with effort that is proportional to the program length, and to avoid an exploding of the amount of enumerative reasoning by using a "divide and conquer" strategy. The aim also is to characterize the progress of the computations. In order to achieve this, a program should be restricted to a set of sequential computations (time-succession of a number of subactions).

Three kinds of constructs aid in viewing a program as the sequential set of computations as implied in the structure principle.

The first of these constructs is concatenation. As shown in Figure 3.1, this construct can be thought of as linear code, parsed and enumerated into a fixed number of subactions, or sequential computational statements. The concatenation construct can be thought of as a single computational statement symbolized by a process box. This process box is often thought of as a "black box" whose most important features are the existance of a single entry and

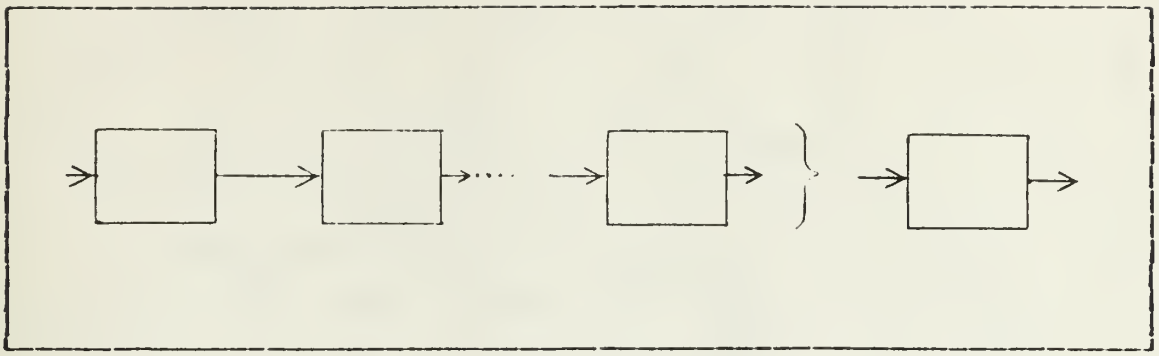

Figure 3.1 Concatenation Construct.

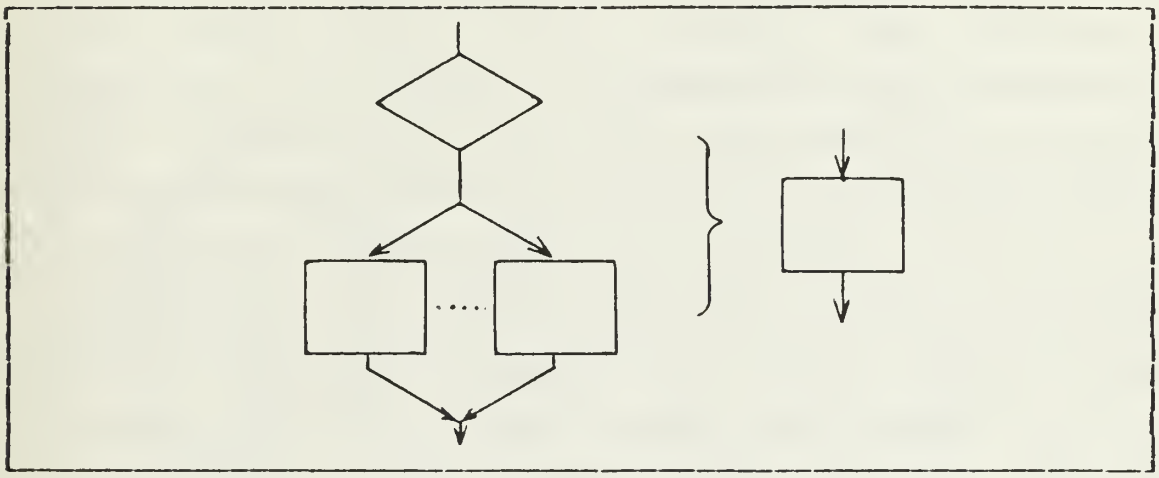

Figure 3.2 Selection Construct.

single exit point and the performance of some function within, the specifics of which are irrelevant.

The second construct, shown in Figure 3.2, deals with selection such as with the IF-THEN-ELSE and CASE mechanisms. The third construct, shown in 3.3, is a looping or repetition construct such as the DO-WHILE, REPEAT-UNTIL or unary decision mechanism. The selection and repetition constructs can also be thought of as a single process box since each construct has only one entry and one exit.

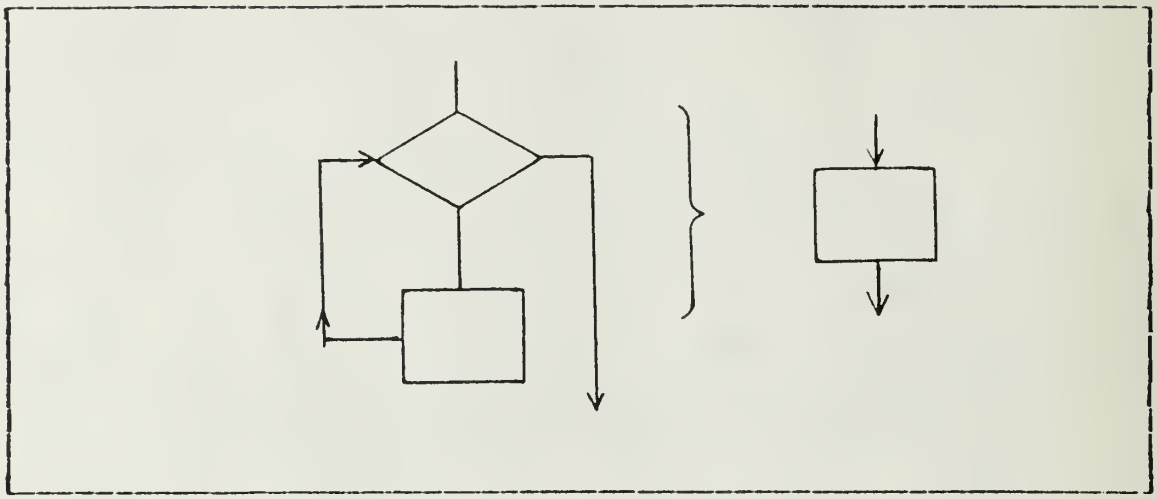

Figure 3.3 Repetition Construct.

Likewise, any program which is composed of concatenation, selection and repetition constructs can be thought of as a single process box itself, with one entry and one exit.

This sequence of transformations can be used as a guide to proving a program correct, and conversely, to design a program in a top-down fashion, starting with a single process box and gradually expanding it to a complex structure built from atomic parts. These transformations can also aid in understanding a program at one level without regard to what happens inside a given construct at another level.

2. Implementation

The theoretical basis of structured programming lends itself to implementation in many of the current programming languages [Ref. 13: p. 148]. All processing in the program must be a single computational statement or a control statement. The control statement is to be a procedure or subroutine call, a selection construct or a repetition construct.

Although these mechanisms are sufficient to write a computer program, some organizations have added extentions. One such extension deals with the tightness of control of the GO-TO statement. For example, if using the GO-TO statement, the programmer must always branch forward in the program, never backwards. Another extention does not allow the GO-TO to branch outside the context of the module in which the GO-TO occurs. [Ref. 13,: p. 152].

Other common programming conventions have been used along with structured programming to further aid program development. These conventions include increasing the readability of the program using formatting techniques such as pretty printing, and increasing the understandability of the program by placing restrictions on the size of the modules. To ensure the integrity of the module, other conventions do nct allow a module to modify any other module. To further ensure integrity, local variables and temporary storage for any given module are not allowed to be shared between modules. The notion of top-down design ensures that the principles of structured programming are firmly embedded within the program.

All these additional corventions, however, are not necessary to support the theory of structured programming, but instead are convenient and aid in augmenting the quality of programming practice.

3. Conversion

Several computer scientists have shown that any computer program could be constructed from combining the basic constructs of concatenation, selection and repetition. However, they ncted that many programmers experienced difficulties since structured programming required a different approach to designing and thinking about programs.

The scientists further posed the question: Could any unstructured program be converted into a structured program? The answer is "no": an arbitrarily selected unstructured program can not be converted into a structured one that performed the same algorithm with the same primitives and no additional variables. Only with the introduction of additional variables (called state variables) is conversion possible. The proofs are lengthy and will not be discussed here, but can be found in [Ref. 17].

F. STRUCTURED AND UNSTRUCTURED PROGRAMS

Just as with most cost-benefit analyses in computer science, there is no accurate way to measure the goodness of a program given a set of criteria. There is no correct or incorrect way of measuring the tradeoffs between the various desirable characteristics of a program. Even given two programs which do the same thing, there is no easy way to compare them with these desirable characteristics.

While there may be no way tc evaluate the "goodness" (or "badness" for that matter) of a program, educated conclusions can be made based on our knowledge of the human IPS, it's capabilities and limitaticns, and these two opposing techniques of designing programs.

IV. SUMMARY

Given the previous discussion, it is now possible to view the value of structured programming in terms that are more closely related to psychological characteristics. The intent of this thesis is to show that, with the current state of knowledge in psychology, it is not possible to build a satisfactory model of the programmer and his/her task. However, one can do better than previous work by identifying the components of such a theory and then based on that define the programming task more clearly.

Two questions were of concern at the beginning of this thesis: First, given the complexity of the psychological issues presented, is the research on the effects of structured programming worthwhile to pursue? Is it worth continuing to study, is it provable, or are there too many variables? The second question asked if there were measurable benefits derived from structured programming methodology. Even if perceived benefits were real, it was not clear they could be quantified or even monitored in order to confirm the effectiveness of structured programming.

To help answer these questions, let us ask: Can a viable model be built which accurately describes the task of understanding the processes of a program?

A. DEVELOPING A MODEL FOR THE UNDERSTANDING TASK

Curtis [Ref. 7: p. 102] proposed steps to develop some aspect of software complexity. From this guideline, the steps to develop a model of the understanding task can be proposed as follows:

1. define (and quantify) the criterion that the metric will be developed to predict
2. propose a model of the processes of the human system which affect this criterion
3. identify the properties of the program which affect the operation of these processes
4. quantify program characteristics
5. validate this model with empirical research

A model of the understanding task implies not only the quantification of the software characteristic, but also a theory of the human system involved. The starting point for developing a way to measure the understandability of a program is 'not an ingenious parsing of software characteristics' as Curtis put it, but an understanding of the human system when it interacts with the program. Of course, the accuracy of this measurement depends on the validity of the assumptions made about the human processes.

1. The Criterion

Claims have been made that the use of structured programming produces cost-effective changes in programming practice. The claims have been broad: its use aids readability, understandability, improves programmer productivity and decreases testing difficulties, to name only a few. The focus of this thesis has been toward understanding the processes of a program, which is an all-encompassing characteristic that appears rudimentary to each of the other claimed benefits. It is for this reason we ask: Does the use of structured programming produce cost-effective changes in the understandability of a program?

2. Evaluation of the Variables

It was established that programming is a human activity. The human is limited by the organization and capabilities of the human memory system. Our attention is limited such that it is very difficult to follow the dynamic structure of long, complex programs. The accuracy of our perception of a program's processes is dependent upon the characteristics of the program and the amount of attention applied. If a program is unorganized and causes a person to keep track of many things at one time, their attention will be distracted and their perception will not be accurate. Therefore, for an unorganized program, the structure does not allow the details of the program to be easily perceived and the program is not as readable.

One's accuracy in perception, strategies in problem solving, and one's knowledge base are factors in understanding. To understand means to relate one thing (here a program) to another (a representation that is in LTM or is created by the human). It does not mean to memorize, but rather to be able to answer questions about the program with limited access to EM (the program listing). Therefore, it also involves the ability to slice. That is to say, the task of understanding always involves the comparison of two things--the one perceived (the program) and a representation in LTM that it can be associated with, either existing or created. The understanding task also involves a relation between these two representations which identifies them as representations of the same thing. A characteristic of understanding is the ability to answer questions about the program's dynamic process based on an understanding of the static program.

Given a specification, the program and its corresponding dynamic process, the programmer's reading task

involves understanding the problem in two different senses. The program must be understood as it relates to the specification and as it relates to its dynamic process when executed on a computer. This correspondence is represented in Figure 4.1.

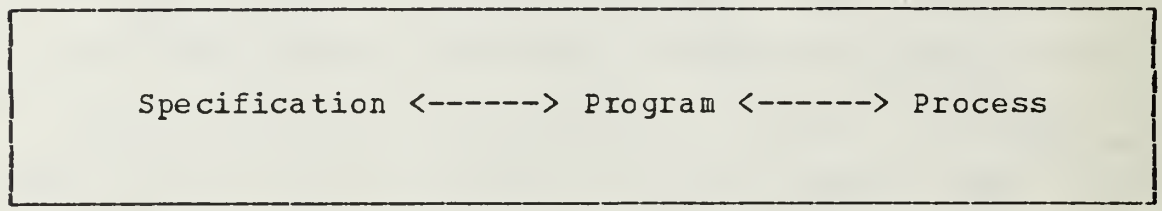

Figure 4.1 Two Senses of Program Understanding.

This task is easier, of course, when the relationship between the specification and the program and the relationship between the program and its dynamic process are simple. In this way, the programmer's task to establish an understanding of the program by creating a relationship between the specification and the process is also simple. "Simple" must be in human terms. For example, the process is the result of the compilation and execution of a program on a computer, which is a well defined and "simple" process mechanically speaking. In the same case, the programmer needs to mentally build a relationship that can be recalled, manipulated and with which assertions about the program can be made. Limited to managing small amounts of information in STM, for programs of any large size it is unlikely that the programmer will be capable of establishing a direct relationship between the program and the process. The programmer must then decompose the program-to-process relationship into many subrelationships. This means that in addition to the understanding of the pieces, the programmer must also have the mechanism tc build understanding of the whole relationship frcm an understanding of the pieces.

3. The Program Properties/Characteristics

We have our limitations. Using the strengths of our facilities, we have devised strategies in program organization and technique that help cvercome our limitations in processing information.

Structured programming employs one such technique. When these mental concepts are applied to structured programming, they imply that structured programming provides a simple, programmer-independent mechanism for hierarchical decomposition of the relationship and combining of subrelationships.

Use of the three branching and control constructs yields a hierarchical decomposition that enforces the structure principle and eliminates much of the randomness and spontaneousness in developing programs. The structure principle helps lessen the conceptual gap between the static program and the dynamic process, making the program appear less complex and easier to decompose. Without this, decomposition rules become numerous and more complex and the program becomes more difficult to understand.

This simpler structure places less of a burden on STM. The programmer is able to better and more easily build an understanding of the pieces, and the relationships necessary to recombine them for an understanding of the whole.

The number of ways the program can be represented is more restricted than the same unstructured program. These constructs force organization and a more readable structure. Fewer, simpler, more readable representations imply it would be easier to develop a relationship with a knowledge structure already constructed and stored in LTM. This implies the structured technique increases the understandability of a program.

4. Quantification of Program Characteristics

With all these seemingly obvious implications, it would appear possible to identify and then measure the psychological variables associated with understanding programs. This is not now possible. Here is why: From our understanding, the programming task involves the task of understanding the process of a given program. Understanding in turn, relies on complex behaviors like reading, decomposing, building, and combining. There are mechanisms, the psychological variables that are the nuts and bolts of the model we wish to build, that link these behaviors with the internal framework of our human processor model--particularly STM, LTM and how information is stored, organized and retrieved from our knowledge base. The psychological variables that would explain these complex behaviors have not even been identified let alone measured. There remains a huge gap between clearly identifying psychological variables and the complex high level behavior we wish to model.

5. Validation of the Model

Even if we could build a plausible psychological model of the understanding task, the nature of the software environment and the influence of individual differences among programmers limit the generalizability of the results from any empirical study. Problems arise when research involves human performance. The dramatic variation in performance among individuals easily disguises any relationship between any software characteristic and its associated criteria. The differences in the performance of some task can often be attributed more toward the differences among programmers performing the task than to differences in the characteristics of the program on which the task is performed.

B. CONCLUSIONS

Results in psychology research have helped to frame the structured programming question in psychological terms. However, the current state of psychology makes it impossible to build a model of the value of structured programming which can be validated through empirical research. Several areas of psychology will require further advancements. While not intended to be an inclusive list, the first has been mentioned above: the need to better understand human performance differences so biases in empirical work can be lessened if not eliminated. Second and closely coupled with this, to gain a better perspective on the burdens placed on LTM more detailed knowledge is needed on how knowledge structures are stored, organized and retrieved into LTM. This would help to define the understanding process more clearly. Thirdly, to provide the basis for all the above, we are in need of both vocabulary and metrics with which to accurately and consistently describe higher level human processes. We are without the very foundation needed to support our assertions. We do not have a vocabulary to describe the decomposition, analysis and composition processes in the understanding task and we have no way to directly observe them. Even if we had this capability, we would need a metric with which to determine the cognitive difficulty (or simplicity) of these processes. We do not even have a metric to define what is "simple". Thus, we do not have the ability to determine whether the cognitive difficulty of structured programming is smaller than that of competing technology. There is also the problem of describing the non-structured programming alternative. It is not sufficient to define non-structured programming as "the absence of structured programming". Rather, it would need to be described in the same terms as structured

programming--that is, the components, such as knowledge base and processes.

While the absence of a complete model of the programming task and associated terminology and metrics is somewhat disappointing, it is this author's subjective opinion that this is the best we can do with the current state of psychology. The missing mechanisms have been identified. Without these, our model of the programming task can not be completed. Furthermore, without a theory, the difficulties with empirical work are to be expected. Improving methodology can not compensate for a lack of theory to suggest these critical variables and how to measure them.

LIST OF REFERENCES

1. Vessey, I. and Weber R., "Research on Structured Programming: An An Empiricist's Evaluation," IEEE Transactions on Software Software Engineering, v. SE-10, pp. 397-407, July 1984.

2. Sheil, B. A., "The Psychological Study of Programming," Computing Surveys, v. 13, pp. 101-119, March 1981.

3. Tracz, W. J., "Computer Programming and the Human Thought Process," Software-Practice and Experience, v. 9, pp. 127-137, 1979.

4. Newell, A. and Simon, H. A., Human Problem Solving, Prentice-Hall, 1972.

5. Norman, D. A., Memory and Attention, An Introduction to Human Information Processing 2e, Wiley and Sons, Inc., 1976.

6. Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information, Psychological Review, v. 63, pp. 81-97, March 1956.

7. Curtis, B., "In Search of Software Complexity," Proc. IEEE Workshop Quant. Software Models for Reliability, Complexity and Cost: An Assessment of the State of the Art, New York, 1979.

8. Simon, H. A., "The Functional Equivalence of Problem Solving Skills," Cognitive Psychology, v. 7, pp. 268-288, 1975.

9. Weiser, M., "Program Slicing," Proceedings of the Fifth International Conference on Software Engineering, pp. 439-449, 1981.

10. Weiser, M., "Programmers Use Slicing When Debugging," Communications of the ACM, v. 25 no. 7, pp. 446-452, July 1982.

11. Dijkstra, E. W., "Go To Statement Considered Harmful," Communications of the ACM, v. 2, pp. 29-32, March 1968.

12. MacLennan, B. J., Principles of Programming Languages, Holt, Rinehart and Winston, 1983.

13. Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall, 1975.

14. Brooks, F. P., Jr., The Mythical Man-Month, Essays on Software Engineering, Addison-Wesley Publishing Company, 1982.

15. Freeman, P., "Fundamentals of Design," IEEE Tutorial on Software Design, Computer Society Press, pp. 2-22, 1983.

16. Wulf, W. A., "A Case Against the GO-TO," Classics of Software Engineering, Yourdon Press, pp. 85-98, 1979.

17. Yourdon, E., ed., Classics of Software Engineering, Yourdon Press, 1979.

BIBLIOGRAPHY

Belady, L.A., "On Software Complexity," Proc. IEEE Workshop Quant. Software Mcdels for Reliability, Complexity and Cost: An Assessment of the State of the Art, New York, 1979.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, W. and Merit, M. J., Characteristics of Software Quality, North Holland 1978.

Brewer, W. F. and Treyens, J. C., "The Role of Schemata in Memory for Places," Cognitive Psychology, v. 13, pp. 207-230, 1981.

Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, 1976.

Dijkstra, E. W., "The Humble Programmer," Communications of the ACM, v. 15, pp. 859-866, October 1972.

Frost, D., "Psychology and Program Design," Datamation, v. 21, pp. 137-138, May 1975.

Simon, H. A., "How Big is a Chunk?," Science, v. 183, pp. 482-488, 1974.

Yourdon, E. and Constantine, L. L., Structured Design: Fundamentals of a Discipline of Computer Program and System Design, Prentice-Hall, 1979.

INITIAL DISTRIBUTION LIST

No.		Copies
1.	Department Chairman Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
2.	Curricular Officer Code 37 Computer Technology Programs Naval Postgraduate School Monterey, California 93943	1
3.	Dr. Gordon Bradley Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, California 93943	3
4.	CAPT Bradford D. Mercer, USAF Code 52Zi Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	LT Cynthia A. C. McGrath, USN 6420 Dorset Drive Alexandria, Va. 22310	1
6.	LCDR Thomas R. McGrath, USN c/o Chief of Naval Operations 163C Washington, D.C. 20370	1
7.	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, California 93943	2
8.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2

Thesis
M188442 McGrath
c.1 Review of the psychological issues relating to the effectiveness of structured programming.